MON: Design and Implementation of Management Overlay Networks for Distributed Systems

Jin Liang, Steven Ko, Indranil Gupta and Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana-Champaign
{jinliang, sko, indy, klara}@cs.uiuc.edu

ABSTRACT

While large-scale distributed clusters such as PlanetLab and TeraGrid are being increasingly used to run complex applications, there is a paucity of software that provides simple *management functions* to developers and deployers of such applications. This paper develops *Management Overlay Networks (MONs)*, that can provide the application deployer the capability (1) to execute *short-term* (on-demand) commands that query or monitor the status of a distributed group of hosts, and (2) to execute *medium-term commands* that enable the persistent running of an application in spite of node failure. The MON mechanisms we develop require minimal operator involvement; further, their simplicity, scalability, lightweight-ness and fault-tolerance enable them to run side-by-side with existing distributed applications. As a proof of concept, this paper presents the design of six commands for on-demand monitoring, as well as a Restart mechanism for persistent application execution. We present experimental results from our implementations running on the PlanetLab cluster, with the data being taken in the days leading up to the SOSP deadline. We also present simulation results for larger system sizes. The concept of MONs addresses concerns and opinions raised by both peer-to-peer networking and Grid computing communities in the recent past.

1. INTRODUCTION

In recent years, large scale, widely distributed computing systems are increasingly being deployed for both commercial and research purposes. Examples are content distribution networks [1], Grid computing systems [11], and the PlanetLab testbed [20]. These systems often consist of large number of computers that are distributed across a wide area network. Such systems suffer from all kinds of failures such as hardware breakdowns, software run-aways, and network connectivity problems [6]. In systems such as PlanetLab and Grid computing networks, failures are the norm rather than the exception.

This leads us to observe that an important component that is missing from such systems today is the capability for an application developer to (1) *monitor* the health of the system, and (2) if necessary, transparently and *automatically adapt* to satisfy application requirements in the face of failures and changing circumstance.

In this paper, we take a first cut at designing, implementing, and evaluating the *Management Overlay Network (MON)*. MON is a distributed management system that contains specific implementations for each of the monitoring and automatic adapting components. The monitoring component of our MON is able to answer simple questions such as "which nodes are currently heavily loaded", "what services are consuming more than 90% of the CPU time", etc. This component can also be used for other purposes such as disseminating code updates, etc.

As a part of the the automatic adapting component of our MON, we implement the *Restart* framework. The Restart framework monitors the health of specific nodes at which a particular application is running. When one or more of these nodes fails, the framework automatically transparently restarts the application at an equivalent number of other non-faulty nodes. We explore the uses of the Restart framework for stateless and soft-state distributed applications such as distributed simulations of peer-to-peer distributed hash tables (p2p DHTs), a popular recent use of PlanetLab. A scientist can thus be assured that the simulation she starts will *always* run on, say, "at least 30 nodes", even if some of the initial nodes crash.

Before we proceed with the details of our design, we take a step back to enumerate the high-level desired characteristics of a MON:

- **On-demand nature:** For the management of a large distributed system, it may not be necessary to constantly monitor the system and alert the human manager upon each failure. Instead, a management system that allows the human manager to instantly query the status of the system may be more desirable.
- **Minimal operator involvement:** Several studies have shown that operator mistakes are a leading cause of system failures [18]. The mechanisms of a MON should be autonomous, and if possible, avoid a human-in-the-loop.

- **Compatibility with existing (legacy) applications:** A MON should ne non-intrusive to existing distributed applications (e.g., legacy applications such as distributed market predictor simulations).

- **Performance:** The mechanisms in a MON should be scalable (as the system size grows), it should be lightweight (involve low message overhead), and it should be resilient to failures during its operation.

The concept of overlay network has been successfully used to build many distributed applications, ranging from file sharing to web caching and real-time media broadcasting. In most existing research work, however, nodes in the overlay must constantly maintain a certain structure among themselves, and repairs must be made if the structure is changed by node arrivals and departures. Maintaining such a structure and satisfying the invariants within it (e.g., for p2p DHTs) all the time can cause unwanted interference and overhead on existing applications.

This precludes us from deploying existing DHTs across the nodes and using it. Rather, we resort to running a lightweight membership protocol that maintains soft-state and structureless membership lists. On-demand MON commands are executed over such a *structureless overlay*.

Since structurless overlays are lightweight, this approach also ties in with our *Performance* goal above. In addition, the Performance goal entails that the mechanisms inside the MON be both *simple* in implementation, while being *effective and reliable*.

The first monitoring component of our MON is implemented through an *on-demand overlay network*. During normal operation time, this MON component maintains no structure among the overlay nodes. Each node merely periodically exchanges partial membership information with other nodes at a low rate. When some management task arrives (e.g., status query or software update), MON can quickly organize the nodes into an appropriate structure (e.g., a tree for status query and a DAG for content distribution), and carry out the management task on this structure. The overlay structure exists as long as the management task lasts (which is usually short), and is discarded as soon as the task finishes.

The second automatic adapting component of our MON is implemented through a Restart framework that needs to run a structureless membership protocol only among the nodes involved in the application.

Seen in a different way, this paper considers only *short-term and medium-term commands* for MONs. The monitoring component of our MON are *short-term commands* issued by the application deployer, while the Restart framework deals with a specific *medium-term command*. Existing centralized management structures (e.g., CoMon [4]) or DHT-based approaches (which we have excluded for reasons cited above), would fall in the cateogry of *long-term commands*.

To summarize, compared with other centralized or decentralized management systems, our on-demand management overlay network has the following advantages: (1) the system is simple and involves very low overhead: Since no structure is maintained during normal operation time, no repair is needed even if there are frequent node failures and recoveries; (2) it is self-organizing and fault resilient: Since nodes periodically exchange membership information with each other, node failures can be detected, and new nodes are discovered; (3) it is adaptive and task specific: When a structure is created on demand, it is based on the current system performance and the task at hand. For example, congested network links can be automatically excluded from the overlay, and different structures can be built for different tasks (e.g., tree for status monitoring and DAG for content distribution).

We have implemented MON and deployed it on about 120 nodes on the PlanetLab. Our extensive experiments on Planetlab, as well as detailed simulation results for larger groups, demonstrates the soundness and utility of MON. We present results from MON monitoring of PlanetLab in the days leading up to the SOSP deadline.

Finally, several prominent Grid researchers such as Ian Foster et al [10] and p2p researchers such as Seltzer et al [16] have called for a convergence of Grid computing and p2p paradigms. The MON project is in tune with this direction, addressing application and infrastructure concerns of both peer-to-peer computing and Grid-like systems.

The rest of the paper is organized as follows. In Section 2, we discuss some previous work about distributed system management. In Section 3, we provide an overview of the MON architecture. Section 4 gives the detailed design of our instant status monitoring, and Section 5 is the design of the restart framework. Next we present evaulation results in Section 6. Finally Section 7 concludes the paper.

2. RELATED WORK

Distributed system management can be divided into two classes, centralized and decentralized. Centralized management systems often involve a management agent running on each device to be managed. These agents can report the status of the device to a central manager node, and control the device based on commands from the manager node. Although such systems are not scalable, they are still used in many existing systems for practical purposes. One example is the grid information service (GIS) [9], where each resource periodically report its status to a central directory server. Users in the system can then query the directory server to obtain status of the system. Another example is the CoMon [4] monitoring system currently deployed on the PlanetLab, where a daemon on each PlanetLab node periodically reports the status of the node to a central server, which then makes the information available through a web interface.

There has also been research work that considered decentralized management system. For example, Amir et al. [7] have considered distributed system management on top of a group communication infrastructure. They considered three kinds of management operations, simultaneous execution, software installation, and consistent network table maintenance. A command issued by a manager node is transmitted to all the nodes by the group communication system. In

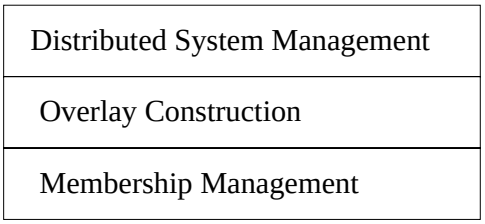

Figure 1: MON Architecture

their system, group communication is implemented as multicast in a local area network. Since multicast is not available in wide area network, and maintaining group membership is difficult in a large system with frequent failures, their system is unlikely to fit the large distributed computing systems that we consider.

Astrolabe [21] and Ganlia [17] are two distributed monitoring system. Both organize distributed computers into a tree hierarchy. Distributed monitoring data are then aggregated and propagated along the tree edges.

Recently, Oppenheimer et al. [19] have considered another decentralized monitoring system (SWORD). In their system, a DHT (Bamboo) is used to store the status information. Each node periodically report their attribute values to the DHT, so that a user or application can query the DHT to find out the nodes of interest. Their system is mainly for resource discovery in a distributed system, therefore, it's not fitted to the system management that we are considering.

There also exist systems for software deployment. CoDeploy [3] is a tool that can deploy software on the PlanetLab. However, it relies on the CoDeeN [2] content distribution network for software dissemination. Therefore, it is not suited for our management system.

3. OVERVIEW OF MON ARCHITECTURE

The architecture of MON is shown in Figure 1. Both components of MON conform to a three layer architecture. At the lowest layer, a membership protocol periodically exchanges membership information with the membership layer of other nodes. This allows each node to build a partial view of the system. This partial view is then used by the upper layer for on-demand overlay construction. Membership exchange also detects node failures, so that failed nodes can be avoided during the overlay construction process.

On top of the membership layer, the overlay construction layer is responsible for constructing an overlay structure among the nodes. The particular overlay structure depends on the application needs (management tasks). For instant status monitoring such as "finding the nodes that are currently most heavily loaded", a spanning tree may be constructed on-demand. For medium term commands such as the restart framework, the overlay is constructed and maintained as long as the application executes. For software upgrade tasks, a more densely connected structure such as a directed acyclic graph (DAG) might be a better choice. If an overlay network is created on-demand, it must be done in a quick fashion. In addition, the construction process should try to include as many live nodes as possible, and ensure that the constructed overlay has good performance such as end to end delay.

The distributed system management layer is concerned with the specific management tasks. For example, how should system status be collected, aggregated and propagated, and how should software data be pushed to the nodes.

4. MONITORING MECHANISMS IN A MON

4.1 Distributed Membership Management

Maintaining an up-to-date global membership list for a large distributed system is difficult, especially when there are frequent node failures and recoveries. In the design of our management overlay network, we have adopted a gossip-style membership management. Specifically, each node mains a partial membership list for the system. This is called the partial view of the system. Periodically, a node picks a random target from its partial view, and sends a `Ping` message to the target. The `Ping` message contains a small number of membership entries, also randomly selected from the partial view. A node receiving a `Ping` message should respond with a `Pong` message, which also includes some random entries selected from the partial view of the `Ping` receiver. The `Pong` message allows the `Ping` sender to estimate its delay from the target. This information can be used for build locality into the on-demand overlay, as we will describe later in Section 4.2.

When a node receives a `Ping/Pong` message, it needs to merge the membership list in the message with its partial view. We assume each partial view has a fixed size. Thus we must decide which membership entries should be dropped, if the partial view is already full. There are different ways to merge the two lists [13]. In our MON design, we use an *age*-based technique to maintain fresh membership entries in the partial view. Specifically, we associate an age with each membership entry. When a node A receives a message from node B, A will create an entry for B, and set its age to 0. When an entry is sent in a `Ping/Pong` message, its age is also included. When a partial view is full, the entry with the largest age is dropped first. Such age-based gossip is effective at eliminating failed nodes from partial views. When a node has failed for some time, its entry in other nodes' partial views is likely to have large age, thus it's likely to be dropped first.

4.2 On-Demand Overlay Construction

The partial views maintained at each node effectively create a densely connected graph among all the nodes. To create an overlay among the nodes is equivallent to creating a spanning subgraph of the partial view graph. In this paper we consider the construction of two kinds of overlays, spanning trees and DAGs. A tree structure is suited for distributed status monitoring and aggregation, and a DAG is suited for software pushing.

To build a spanning tree, an initiation node randomly selects k nodes from its local view, and sends a `Create` message to each of them. If a node receives a `Create` message for the first time, it will respond with a `CreateAck` message and become a child of the `Create` sender. It will also randomly

pick k nodes from its own partial view, and send the `Create` message to them. If a node receives a `Create` message for a second time, it will respond with a `Prune` message, because it is already in the tree. It has been shown that assuming the partial views represent uniform sampling of the system, such tree construction will cover all the nodes with high probability, if $k = \Omega(\log N)$, where N is the total number of nodes in the system [14].

One problem with the above tree construction algorithm is that the created overlay is not locality aware, which means messages between parents and children may need to traverse wide area networks. We can use a simple technique to improve the locality of the overlay. Whenever we need to select k nodes from the partial view, we will randomly select $k + c$ nodes, and sort the nodes according to their distance from the current node. The c nodes that have the largest delays are then discarded. This ensures at each hop, no extremely far-away nodes are selected. However, this may introduce another problem. Since we are no longer selecting nodes uniformly at random, the probability that all nodes are covered (called the *coverage*) is reduced. To improve the coverage of the overlay, if a node receives one or more `Prune` messages, it can try to send the `Create` message to as many as d additional nodes.

The above process, each node only accepts one parent, thus the resulting structure is a spanning tree. If a node accepts more than one parent, a DAG can be created. A DAG is preferable to a tree for software pushing tasks, because a node can download different parts of the software from different upstream nodes (parents), thus free from the bottleneck link problem with a tree structure.

To avoid loops when creating a DAG, each node is assigned a *level*. When a node sends a `Create` message, it includes its level l in the message. A node receiving a `Create` message for the first time will set its level to be $l+1$, where l is the level included in the message. Thereafter, additional parents are accepted only if the level is smaller than the current node.

4.3 Instant Status Monitoring

As explained in the Introduction, an operator or application developer may need to query the status of a system, in order to assess the health of the system and diagnose failures. Our management overlay network supports such instant status monitoring by dynamically creating an overlay among the distributed nodes and execute the status monitoring commands on the overlay.

We have implemented the following status monitoring commands in our MON implementation.

- `Count`
- `Depth`
- `Topology`
- `Filter`
- `LoadAverage`
- `LoadHistogram`

All these commands are executed in a similar fashion. First, a client side program sends the command to a nearby MON node. The node will execute the command locally, and produce some local results. The node will also send the command down to its children, and aggregate its local result with the data returned by its children. When a node has received data from all children, it will send the aggregate data back to its parent.

The first three commands are related to the management of the overlay structure itself. The `Count` command is used to count the number of nodes in the tree. The local execution just produces the number 1. And aggregation means to sum up the numbers from all children and the local execution. Since the local execution is so simple, we can measure the time taken for a `Count` command and use it as a baseline performance measure of the overlay structure.

Similarly, the `Depth` command returns the depth of a tree or DAG (i.e., the maximum level of a node). The local execution just produces the level of the local node. Aggregation means selecting the maximum level returned by a child, or the local level if the node has no children.

The `Topology` command returns the topology of the spanning tree. At each node, if it is a leaf node, it just sends an empty message to its parent. Otherwise, it will build a list of the edges from itself to its children, merge this list with the lists received from its children, and send it to its parent.

The `Filter` can be used for different status monitoring purposes. We assume the goal of the status monitoring is to find out nodes of interest (e.g., nodes with excessively high or low load). Thus the local execution of the command involves evaluating a filter statement. If the evaluation result is false, it means the local node should be excluded in the monitoring result. If it is true, then some filter specific data is returned as the local execution result. In order to keep the overlay management framework free from the specifics of individual filters and to make the status monitoring extensible, we use an external evaluation engine for the local execution of the status monitoring command. Our current evaluation engine supports simple filter statements that can be expressed in the form `<resource> <op> <value>`, where `<resource>` could be CPU load average, free memory, or disk space usage. `<op>` could be greater than or less than. Note there are many data sources on the PlanetLab, such as the CoMon and Ganglia daemons. Our evaluation engine can be easily integrated with such data sources and support more complex status queries.

Finally, the `LoadAverage` and `LoadHistogram` are especially for our status monitoring experiments on the PlanetLab. `LoadAverage` returns the total load in a subtree, and the number of nodes in the subtree, so that the parent node can compute the average node. `LoadHistogram` returns the distribution of CPU load for different subtrees, and a node will merge the histogram from its children and produce a new histogram to be sent to its parent.

In the above we have focused on the utilization of on-demand overlay networks for instant status monitoring. For a large distributed system, we may also need to distribute content

such as a new software updates to all the nodes. Traditionally such content distribution has been realized by reliable multicast. However, a multicast tree may suffer from a badly chosen overlay link. This means the bandwidth achieved by a multicast tree is limited by its bottleneck link. Recently, it has been realized that to efficiently utilize the available bandwidth of different nodes, each node should be allowed to download data from multiple parents [8, 15]. Our management overlay network can support the efficient content distribution by building a DAG structure among all the nodes. However, precisely how data is disseminated among the DAG structure, for example, whether a pull or push model should be used, and how should flow control be implemented, is beyond the scope of this paper.

5. THE RESTART FRAMEWORK

5.1 Overview

Our *Restart framework* provides a convenient way to manage certain kinds of applications on large-scale distributed systems. The goal of the *Restart framework* is simple - it tries to maintain the same number of application copies running on a distributed system. By doing this, it minimizes the neceessity of human intervention to manually restart a copy of an application on a different machine, especially when a host crashes. The *Restart framework* thus not only provides transparency, it is also application-independent. Thus, it is not necessary to modify the existing applications to be able to use the *Restart framework*.

The *Restart framework* consists of two functions - detecting crashes and restarting a copy of the target application. To detect crashes, each node sends periodic heartbeats to its "neighbors" (see Section 5.3 for the definition). When a node has not received any heartbeats from any of its neighbors for a certain amount of time, it starts the process of restarting a copy of the application.

5.2 Target Applications

The *Restart framework* mainly supports stateless and soft-state applications that have a dynamic join/leave algorithm. Many of the popular distributed applications fall into this category, including DHTs and several Grid applications. DHTs are soft-state; each node constructs a routing table, which is usually refreshed periodically by either heartbeats or ping-pongs. Also, almost all DHTs have a join/leave algorithm, as they provide resiliency against dynamic stresses such as churn. Grid applications usually have an algorithm to deal with dynamic node stresses as well.

Thus, it is particularly useful for a research or a scientist who wants to run her experiment with DHT-based or Grid-based applications for a relatively long period of time. It automatically restarts a new copy of the application of her experiment without her intervention in the face of a node crash. Many of the current experiments on large-scale distributed systems (e.g. PlanetLab) are based on DHT applications, and thus can be benefited by the *Restart framework*.

However, the *Restart framework* does not yet support stateful applications, since it does not store any application-specific states. For stateful applications, techniques based on checkpointing and replication can be used to provide similar functions as the *Restart framework* does.

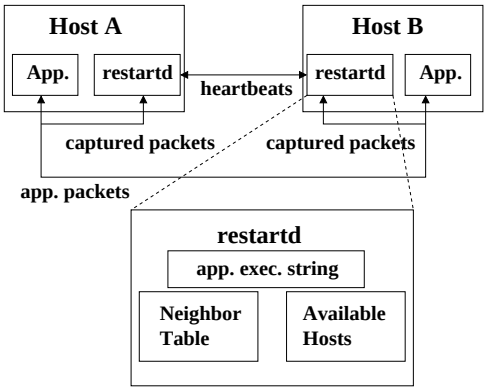

Figure 2: The architecture of the *Restart framework*

5.3 Design

Figure 2 shows the architecture of the *Restart framework*. *restartd* is a daemon running on every host. It sends periodic heartbeats to other daemons on different hosts for crash detection. As mentioned earlier, the *Restart framework* consists of two functions, detecting crashes and restarting. We first explain how to detect crashes, and how to restart next.

5.3.1 Detection

We exploit the fact that every distributed application forms an overlay by itself. Any pair of nodes that exchange application packets form an overlay link. In other words, no extra overlay links are added, beyond those used by the application.

restartd at each node sends periodic heartbeats to the neighbors of the application that it supports. A heartbeat message contains the list of the sender's neighbors. Thus, each node knows every other node in its two-hop distance. When a node has not received any heartbeats from one of its neighbors for a certain period of time, the node decides that the neighbor has crashed and this list is used to notify all the neighbors of the crashed node. Note that this is only one possible approach for detecting failure, since it is possible that a node may be reachable from one node but not reachable from others. More sophisticated algorithms exist in the literature (e.g., [12]).

Alternatively, one could use the application packets themselves to detect crashes. This approach has the advantage of saving resources such as processing power and network bandwidth. However, it is generally difficult to tell whether a host has crashed or not just by monitoring traffic, without any knowledge about the application.

As shown in Figure 2, *restartd* uses passive monitoring to discover new neighbors by capturing packets going through the port of the application. Thus, a group of restartd daemons does not try to form any structure by themselves actively. Rather, they are formed into approximately the same overlay of the application by passively monitoring the traffic of the application. Hence, *restartd* and the application on a same host have the same group of neighbors.

5.3.2 Restart

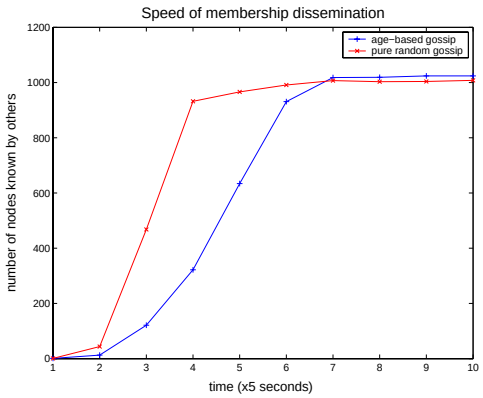

Figure 3: Membership dissemination

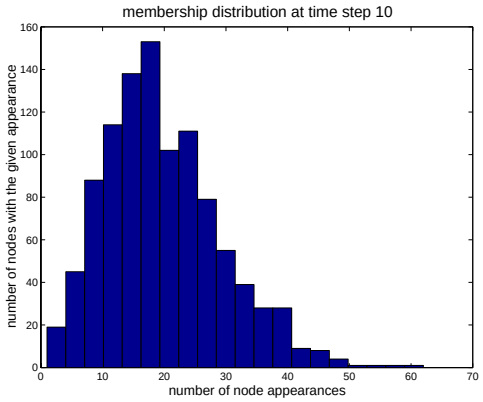

Figure 4: Membership distribution

After a node detects that one of its neighbors has crashed, it notifies all the neighbors of the crashed node. Upon receiving this notification, each neighbor calculates the numerical difference between its IP address (as a 32-bit integer) and that of the crashed node. The neighbor with the smallest difference (or the smallest IP address, if there is a tie) will receive in charge of starting a new copy of the application. Since each node knows all the nodes within a two-hop distance, this activity does not require any additional messages. Also, this activity tends to balance the responsibility of starting new copies across the nodes.

To start a new copy of the application, *restard* uses a list of available hosts and a string of application execution command, obtained out-of-band. The rest of the process is simple; *restard* tries to start a copy on a host in the list one by one until it succeeds.

6. EVALUATION

We have implemented MON in C++. Our implementation consists of consists of about 4000 lines of C++ code. The relatively small code size is the result of our simple system design. In this Section, we present both simulation results and experiment data obtained from a 120-PlanetLab deployment of MON.

6.1 Membership Simulation Results

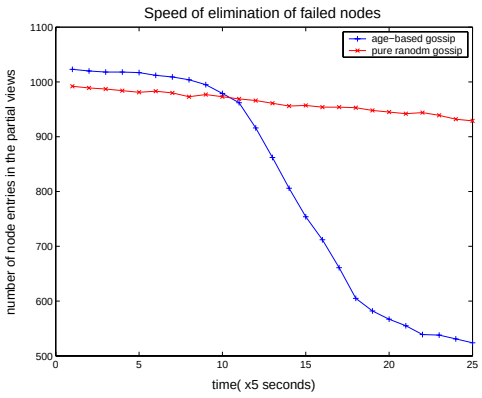

Figure 5: Failure nodes phaseout

We first evaluate the performance of the membership protocol. Figure 3 and Figure 5 show the ability of our membership protocol to quickly disseminate membership information and eliminate failure nodes. Figure 3 shows that in a network of 1024 nodes, initially at time 0, every node knows about only one common node. Starting from time 0, each node begins to periodically exchange membership information, either use our age-based gossip or pure random gossip (node entries are randomly selected for dropping, when the partial view is full). For both protocols, the gossip period is 10 seconds, which means on average each node will send one `Ping` message every 10 seconds. The partial view at each node contains at most 20 entries, and each `Ping/Pong` message contains at most 10 entries. We measure the membership distribution twice every protocol period, or every 5 seconds. Figure 3 shows that both protocols disseminate the membership information quickly. It takes about 3 protocol periods, for every node to be known by some other nodes, although pure gossip is slightly faster than age-based gossip.

Figure 4 shows the distribution of the membership information for the age-based gossip protocol at time step 10. We can see that most nodes are known by up to at most 60 nodes.

Figure 5 shows how fast the membership protocols can eliminate failed nodes. At time 0, we randomly kill 50% of the nodes, and measure the number of different nodes in the partial views every five seconds. The figure shows that initially, because the entries for the failed nodes are still young, the number of different entries decrease slowly. However, after about five protocol periods, the entries for the failed nodes become old enough, and are quickly removed from the system. In contrast, purely random gossip cannot effectively detect node failures, and the entries for most failed nodes persist in the partial views for a long time.

6.2 Performance of Overlay Construction

We have implemented the on-demand overlay construction and status monitoring component and deployed it on about 120 PlanetLab nodes. In this subsection, we present experiment results to evaluate the performance of our on-demand overlay construction algorithms. We focus on the construction of trees, and our metric of interest is the coverage and performance of the overlay, i.e., the nubmer of live nodes

Table 1: Performance of tree construction algorithms

	Coverage	Delay (ms)	Tree Height
Random	23.51	545.51	4.93
Locality ($c = 3$)	23.11	539.10	4.91
Additional ($d = 2$)	23.72	465.36	4.81

Table 2: Performance for 115 node network

Coverage	Delay (ms)	Tree Height	Construct Time(ms)
107.86	645.10	7.48	1664.78

covered, and the end to end delay from the root to the leaves and back. We also present some interesting data about the PlanetLab collected using our MON deployment.

Table 1 shows the performance of the algorithms for an experiment on PlanetLab. We deployed the management overlay network on 25 PlanetLab nodes, and periodically create a tree overlay among the nodes, and execute the `Count` command. The table shows the average number of nodes covered in an overlay, and the time spent for the `Count` command. We also measure the depth of the tree (the largest level of a leaf) using the `Depth` command. The first row is for pure random child selection. When a node receives a `Create` message, it randomly selects k children, without considering locality of the overlay. k is set to 4 in our experiments. The second row is for locality aware overlay construction. Each node selects $k + c$ ($c = 3$ for our experiments) nodes and remove the c nodes with the largest delay. The last row shows that to improve the coverage, we let each node try as many as d ($= 2$) other nodes, if they receive one or more `Prune` messages from the initial k nodes.

Table 1 shows that for pure random child selection with $k = 4$, on average 23.51 nodes out of the 25 are covered, and a `Count` command takes 545.51 ms. For locality aware overlay construction, the delay reduces to about 539ms, but the coverage also decreases. If we use locality aware construction, but allow nodes to try additional children when they receive `Prune` messages, we not only achieve a higher coverage of 23.72, but achieve a delay that is about 80ms smaller than pure random selection. This is because when we allow each node to find more children, the tree becomes shallower, thus the end to end delay involves fewer message hops.

Table 2 shows the performance of the tree construction algorithm for a network of 115 nodes. The number of children k is 6, the locality entries c is 3, and the additional entries for coverage is 3. We can see that for a relatively large network, on average our management overlay network can construct an overlay tree dynamically in less than 2 seconds, and execute a simple `Count` command less than 650 ms. We believe these are encouraging results, especially considering that these results were obtained in the days before the SOSP deadline, when many of the Planetlab nodes were heavily loaded. In fact, Figure 6 shows the cumulative distribution function (CDF) of the response time for the `Count` command. We can see that some times the command is finished in as small as 317 ms. The reason that most executions take about 700 ms is that the network is heavily loaded at the

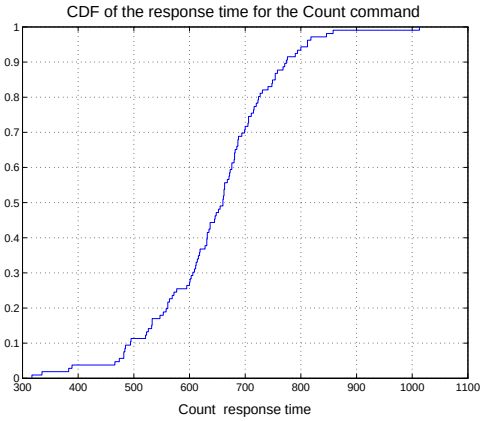

Figure 6: CDF of response time for the Count command on the 115 node network

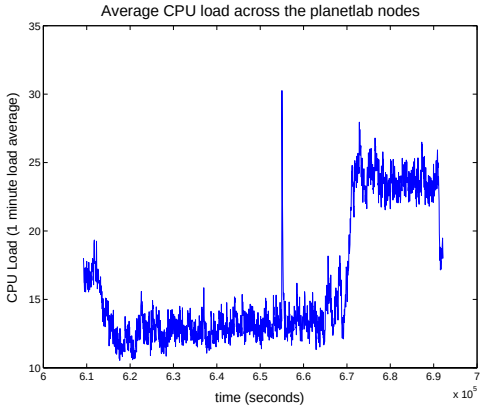

Figure 7: Average load on about 120 planetlab nodes from 2:20:36pm, March 23, for about 23 hours

time of our experiments.

Now we look at some of the status monitoring data we have collected from the PlanetLab using out MON deployment, just before the SOSP deadline. Every 60 seconds, we use a script to connect to a nearby MON node, create a tree among all the nodes, and use the `Filter` command to find out the most heavily loaded nodes and the most lightly loaded nodes. The average load on all the nodes is also computed.

Figure 7 shows the average load on the 120 nodes for about 23 hours. The figure shows that throughout the monitoring period, overall the planetlab nodes are heavily loaded, the average CPU load on all nodes is above 10 all the time, and can be as high as 25. For most of the time, the average load does not vary much, this is probably due to the long running nature of most PlanetLab experiments. However, there could be occasional load spikes, or sudden jumps. These are probably caused by the start and stop of large scale experiments.

In our experiments, each time an overlay is created, we use a `Filter` command to find out the nodes that are heavily

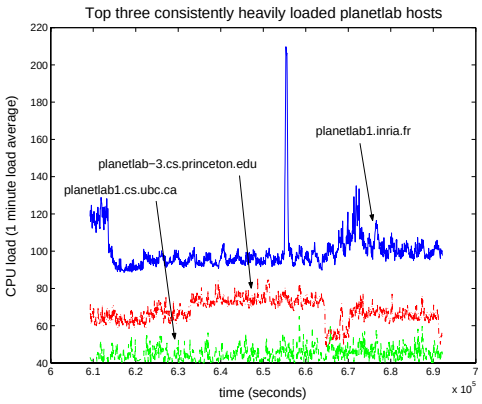

Figure 8: Top three consistently heavily loaded nodes

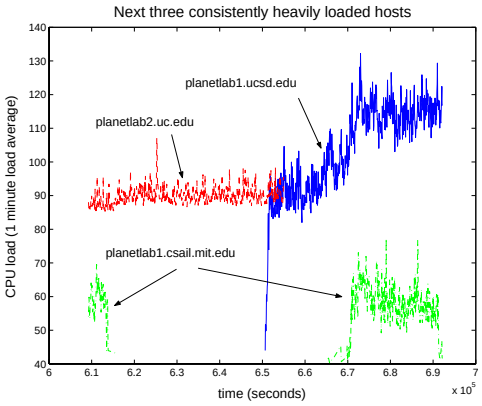

Figure 9: Next three consistently heavily loaded nodes

loaded (with a CPU load greater than 40) and those that are lightly loaded (with a CPU load less than 2). Through out the monitoring period, we found that a few nodes are consistently heavily loaded. Figure 8 shows the top three such nodes. The load on `planetlab1.inria.fr` is consistently higher than 85, and the three machines together contribute to more than 10% of the overall load on the system.

Figure 9 shows the load on the next three consistently loaded nodes. One interesting thing is that the peak load on these machines are not necessarily lower than the top three nodes, but they are only heavily loaded for part of out monitoring period. This is probably due to the start/stop of large scale experiments. It also means that most of the load on the machines might have been caused by a few distributed experiments.

In contrast to the heavily loaded nodes, some nodes are consistently very lightly loaded. This means the load on PlanetLab is very unevenly distributed. In fact, Figure 10 shows a snapshot of the load distribution on the PlanetLab. We can see only a small number nodes have CPU load larger than 20. More than half of the nodes are having a load of less than 10.

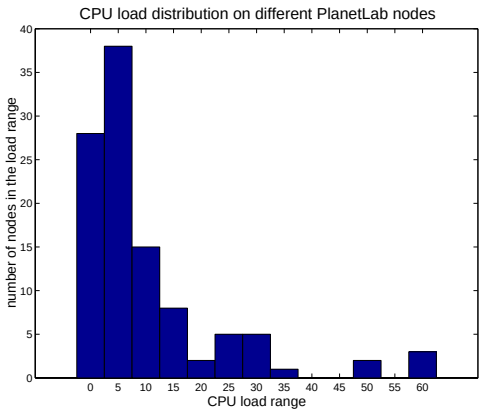

Figure 10: CPU load distribution on different nodes

6.3 Restart Framework

The *Restart framework* is also implemented in C++ and deployed on 43 PlanetLab nodes on the day of SOSP deadline. Libpcap packet capture library is used to monitor the traffic of applications. 23 nodes are used as *available hosts*, available to run a new copy of an application when a node crashes. Initially, only 20 nodes run an application. For our experimenets, we choose FreePastry [5] as our application supported by the *Restart framework*.

Since there are only 43 nodes in total that might run FreePastry, they are all contained in the routing table of each other by the Pastry algorithm. In other words, the overlay topology is a full mesh. In this experiment, the goal of the *Restart framework* is to have at least 20 running FreePastry nodes persistently with a just brief period of disruption.

The following values are used across all the experiments.

- *Heartbeat interval* : This is the interval between consecutive heartbeats that *restartd* sends (1 second).
- *Grace period* : If *restartd* has not received any heartbeats during this amount of period from a node, it decides that the node has crashed (15 seconds).

Figure 11, 12, 13, 14 show the number of active FreePastry nodes over time. It is measured by a script that tries to open the port that FreePastry uses. If it succeeds, the node is counted as alive. In fact, Figure 11 through Figure 14 all show a small degree of fluctuation, even without manual node crashes; this is because heavy network traffic makes the alive counting unreliable.

To emulate host crashes, another script is used to kill some nodes manually. Again, this approach does not guarantee the actual process termination. However, we have not observed any anomalous behavior in this regard during the experiment.

Figure 11 shows the number of active nodes over time. A node is killed at time 20, and the *Restart framework* generates a new node at time 42. Considering that the *grace period* is 15 seconds, roughly around 7 seconds are required

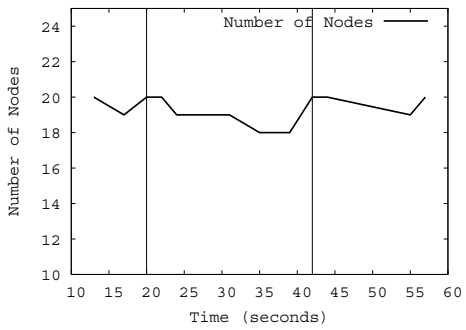

Figure 11: The result from a PlanetLab experiment with one crash. The plot shows the number of nodes over time. One node is killed at time 20 (indicated by a vertical line). The *Restart framework* starts a new node at time 42 (also indicated by a vertical line).

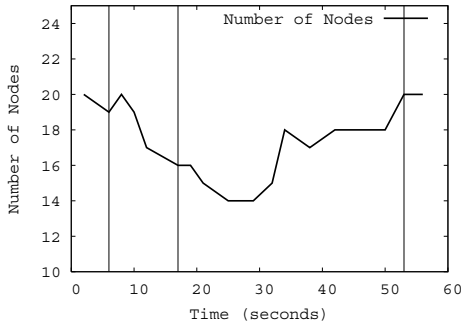

Figure 12: The result from a PlanetLab experiment with 5 crashes. The plot shows the number of nodes over time. 5 nodes are killed during the period of time 5 to 17 (indicated by vertical lines). The *Restart framework* completely generates 5 new nodes at time 53 (also indicated by a vertical line).

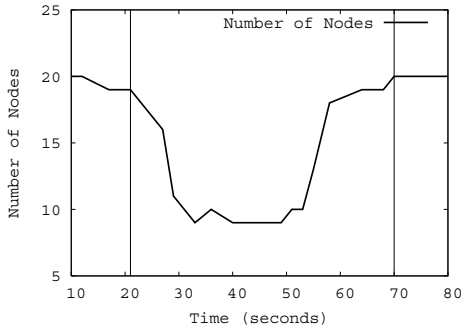

Figure 13: The result from a PlanetLab experiment with 10 crashes. The plot shows the number of nodes over time. 10 nodes are killed almost at the same time at time 21 (indicated by a vertical line). The *Restart framework* completely generates 10 new nodes at time 70 (also indicated by a vertical line).

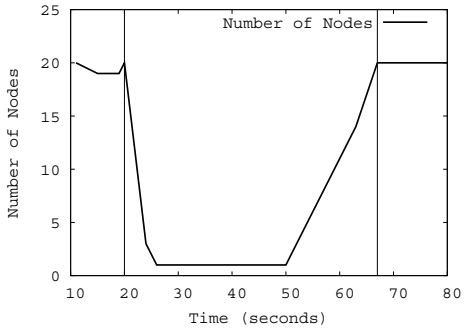

Figure 14: The result from a PlanetLab experiment with 19 crashes. The plot shows the number of nodes over time. 19 nodes are killed almost at the same time at time 20 (indicated by a vertical line). The *Restart framework* completely generates 19 new nodes at time 67 (also indicated by a vertical line).

Table 3: Average recovery time of PlanetLab experiments. Each experiement is repeated 5 times.

	Time
1 node	20.5 seconds
5 nodes	30.39 seconds
10 nodes	35.79 seconds
19 nodes	52.6 seconds

additionally to generate a new FreePastry node. Figure 12 shows a similar plot, but 5 nodes are manually killed instead of 1 node. It takes about 11 seconds (from 6 to 17) to kill 5 nodes manually, and 5 new nodes are completely generated at time 53. Thus, it takes about 36 seconds to finish generating 5 new nodes. Figure 13 shows the similar plot with 10 nodes. The recovery time takes about 50 seconds. Lastly, Figure 14 shows the similar plot with 19 nodes. Here, we kill every node except the bootstrap node of FreePastry. It also takes about 50 seconds to recover from massive failures, similar to the case with 10 crashed nodes.

Table 3 shows the average recovery time with various number of node crashes. Each experiment is repeated 5 times to calculate average recovery time. It shows that it takes less than a minute to completely recover from failures even when almost all nodes are crashed.

7. CONCLUSION

We have designed and implemented a management overlay network (MON). Central to MON is the idea of (1) on-demand overlay construction, where nodes are dynamically organized into appropriate structures; and (2) the restart framework for persistent execution of distributed applications. We believe the idea is especially suited to the management of large, distributed computing systems.

8. REFERENCES

[1] Akamai. http://www.akamai.com/.

[2] CoDeeN content distribution network. http://codeen.cs.princeton.edu/.

[3] CoDeploy software deployment tool. http://codeen.cs.princeton.edu/codeploy/.

[4] CoMon planetlab statistics. http://codeen.cs.princeton.edu/comon/.

[5] FreePastry. http://freepastry.rice.edu/.

[6] R. Adams. Distributed system management: Planetlab incidents and management tools. PlanetLab Design Notes PDN-03-015.

[7] Y. Amir, D. Breitgand, G. V. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *Third International Workshop on Services in Distributed and Networked Enviroments (SDNE'96)*, June 1996.

[8] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *In Proceedings of ACM SIGCOMM 2002*, 2002.

[9] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *The 10th IEEE Symposium on High Performance Distributed Computing (HPDC10)*, 2001.

[10] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS 03)*, Feb. 2003.

[11] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 2001.

[12] I. Gupta, T. Chandra, and G. Goldszmidt. On scalable and efficient distributed failure detectors, 2001.

[13] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *Middleware 2004*, 2004.

[14] A.-M. Kermarrec, L. Massoulie, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(2), February 2003.

[15] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP'03*, October 2003.

[16] J. Ledlie, J. Shneidman, M. Seltzer, and J. Huth. Scooped, again. In *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS 03)*, Feb. 2003.

[17] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30, July 2004.

[18] K. Nagaraja, F. Oliveira, R. Bianchinia, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in Internet services. In *Proc. Operating Systems Design and Implementationj (OSDI)*, pages 61–76, 2004.

[19] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Scalable wide-area resource discovery. Technical report, University of California at Berkeley, July 2004.

[20] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *1st Workshop on Hot Topics in Networks (HotNets-I)*, Princeton, New Jersey, 2002.

[21] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalabel technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.